

ȘTEFAN TANASĂ, ȘTEFAN ANDREI, CRISTIAN OLARU

Java

de la 0 la expert

Ediția a II-a
revăzută și adăugită

POLIROM
2011

Seria *Web* este coordonată de dr. Sabin Buraga
(Facultatea de Informatică, Universitatea „Al.I. Cuza”, Iași)

© 2003, 2007, 2011 by Editura POLIROM

Această carte este protejată prin copyright. Reproducerea integrală sau parțială, multiplicarea prin orice mijloace și sub orice formă, cum ar fi xeroxarea, scanarea, transpunerea în format electronic sau audio, punerea la dispoziția publică, inclusiv prin internet sau prin rețele de calculatoare, stocarea permanentă sau temporară pe dispozitive sau sisteme cu posibilitatea recuperării informațiilor, cu scop comercial sau gratuit, precum și alte fapte similare săvârșite fără permisiunea scrisă a deținătorului copyrightului reprezintă o încălcare a legislației cu privire la protecția proprietății intelectuale și se pedepsesc penal și/sau civil în conformitate cu legile în vigoare.

www.polirom.ro

Editura POLIROM

Iași, B-dul Carol I nr. 4; P.O. BOX 266, 700506

București, Splaiul Unirii nr. 6, bl. B3A, sc. 1, et. 1, sector 4,

040031, O.P. 53, C.P. 15-728

Descrierea CIP a Bibliotecii Naționale a României:

TANASĂ, ȘTEFAN

Java : de la 0 la expert / Ștefan Tanasă, Ștefan Andrei, Cristian Olaru. – Ed. a II-a rev. –
Iași : Polirom, 2011

ISBN : 978-973-46-2405-8

I. Andrei, Ștefan

II. Olaru, Cristian

004.43 JAVA

Printed in ROMANIA

Cuprins

<i>Cuvânt înainte</i> (Dan Grigoraș)	15
<i>Prefață la ediția a II-a</i>	17
1. INTRODUCERE	19
1.1. Cuvinte-cheie	19
1.2. Istoricul și caracteristicile limbajului Java	20
1.3. Instalare	21
1.4. Tipuri de aplicații Java	23
1.5. Mașina virtuală Java	29
1.6. Concluzii	31
1.7. Test-grila	32
1.8. Exerciții propuse spre implementare	32
2. FUNDAMENTELE LIMBAJULUI JAVA	33
2.1. Cuvinte-cheie	33
2.2. Fișiere-sursă	34
2.3. Atomi lexicali	35
2.3.1. Caractere Unicode	36
2.3.2. Traduceri lexicale	37
2.4. Tipuri de date	39
2.5. Expresii și operatori	46
2.6. Variabile	64
2.7. Declarații și inițializări	69
2.8. Conversii	72
2.8.1. Contextele de conversie	76
2.9. Structuri de control	82
2.9.1. Instrucțiunea <code>void</code>	83
2.9.2. Instrucțiunea etichetă	83
2.9.3. Instrucțiunea expresie	84
2.9.4. Instrucțiunea <code>if</code>	84
2.9.5. Instrucțiunea <code>switch</code>	85
2.9.6. Instrucțiunea <code>while</code>	88
2.9.7. Instrucțiunea <code>do</code>	90
2.9.8. Instrucțiunea <code>for</code>	92
2.9.9. Instrucțiunea <code>break</code>	96

14.4.3. Fișierul Map	820
14.4.4. Fișierul cuprins	821
14.4.5. Fișierul index	823
14.4.6. Indexarea și căutarea	824
14.4.7. Compresia și încapsularea	827
14.5. Atașarea help-ului aplicațiilor	827
14.5.1. Clasa HelpSet	828
14.5.2. Clasa HelpBroker	829
14.5.3. Clasa CSH	831
14.5.4. Adăugarea help-ului ferestrelor	832
14.5.5. Adăugarea help-ului componentelor	833
14.5.6. Activarea help-ului prin apăsarea butoanelor	833
14.5.7. Help activat folosind mouse-ul	834
14.5.8. Scufundarea help-ului în aplicație	835
14.6. Concluzii	836
14.7. Test-grilă	836
14.8. Exerciții propuse spre implementare	837
14.9. Proiecte propuse spre implementare	837
15. INTERNAȚIONALIZAREA APLICAȚIILOR	839
15.1. Cuvinte-cheie	839
15.2. Introducere	840
15.3. Codări	841
15.3.1. Despre Unicode	841
15.3.2. Seturi de caractere	842
15.4. Setarea localizării	845
15.5. Separarea datelor	848
15.6. Formatări	851
15.6.1. Formatarea numerelor	852
15.6.2. Reprezentarea monetară	852
15.6.3. Formatarea datei calendaristice și a orei	853
15.6.4. Formatarea mesajelor	853
15.7. Lucrul cu fonturi	855
15.8. Introducerea textului	856
15.9. Pașii necesari pentru internaționalizarea aplicațiilor	857
15.10. Concluzii	858
15.11. Test-grilă	858
15.12. Exerciții propuse spre implementare	860
15.13. Proiecte propuse spre implementare	860
<i>Bibliografie</i>	861

2.9.10. Instrucțiunea continue	97
2.9.11. Instrucțiunea return	98
2.9.12. Instrucțiunea throw	99
2.9.13. Instrucțiunea synchronized	101
2.9.14. Instrucțiunea try-catch-finally	102
2.10. Concluzii	107
2.11. Test-grila	109
2.12. Exerciții propuse spre implementare	115
2.13. Proiecte propuse spre implementare	116
3. CLASE, INTERFEȚE ȘI TABLOURI	119
3.1. Cuvinte-cheie	119
3.2. Clase	120
3.2.1. Domeniul de vizibilitate al numelui unei clase	121
3.2.2. Modificatorii unei clase	123
3.2.3. Clase derivate	126
3.2.4. Clasa Object	127
3.2.5. Implementarea interfețelor	130
3.2.6. Declarațiile membrilor unei clase	133
3.2.6.1. Niveluri de acces	135
3.2.6.2. Declarațiile atributelor unei clase	140
3.2.6.3. Declarațiile metodelor unei clase	151
3.2.7. Inițializatori statici	163
3.2.8. Declarațiile constructorilor	163
3.2.9. Clase interioare	169
3.2.10. Clase generice	173
3.2.11. Enumerări	176
3.2.12. Erori și excepții	180
3.2.13. Distrugerea obiectelor și eliberarea memoriei	183
3.2.14. Divizarea unei aplicații în fișiere	187
3.3. Interfețe	190
3.3.1. Declarațiile atributelor unei interfețe	191
3.3.2. Declarațiile metodelor unei interfețe	194
3.3.3. Moștenire multiplă prin intermediul interfețelor	196
3.4. Tablouri	201
3.5. Conversii ale tipului referință	209
3.5.1. Conversii implicite ale tipului referință	209
3.5.2. Conversii explicite ale tipului referință	211
3.5.3. Conversiile implicite de la tipurile primitive la clasele wrapper	217
3.5.4. Conversiile implicite de la clasele wrapper la tipurile primitive	218
3.6. Concluzii	218
3.7. Test-grila	219
3.8. Exerciții propuse spre implementare	239
4. ȘIRURI ȘI STRUCTURI DINAMICE DE DATE	243
4.1. Cuvinte-cheie	243
4.2. Șiruri de caractere	244

4.2.1.	Interfața CharSequence	244
4.2.2.	Clasa String	245
4.2.3.	Clasa StringBuffer	251
4.2.4.	Clasa StringTokenizer	255
4.3.	Structuri dinamice de date	257
4.3.1.	Interfața Enumeration	257
4.3.2.	Iteratori	258
4.3.3.	Colecții	259
4.3.4.	Tabele de asocieri	260
4.3.5.	Compararea obiectelor	264
	4.3.5.1. Interfața Comparable	264
	4.3.5.2. Interfața Comparator	265
4.3.6.	Mulțimi	267
4.3.7.	Liste	268
4.3.8.	Clasa BitSet	273
4.3.9.	Clasa Properties	275
4.4.	Concluzii	280
4.5.	Test-grila	280
4.6.	Exerciții propuse spre implementare	281
5.	FIȘIERE, FLUXURI DE DATE ȘI SERIALIZAREA OBIECTELOR	283
5.1.	Cuvinte-cheie	283
5.2.	Introducere	284
5.3.	Clase Java pentru intrări și ieșiri	286
	5.3.1. Clasa File	287
	5.3.2. Clasa RandomAccessFile	289
	5.3.3. Clase Java pentru fluxuri de intrare/ieșire	292
	5.3.3.1. Fluxuri de nivel scăzut	293
	5.3.3.2. Fluxuri filtru de nivel înalt	296
	5.3.3.3. Fluxuri pentru citire și fluxuri pentru scriere	299
	5.3.3.4. Clasele PrintWriter și BufferedReader	301
	5.3.3.5. Clasa StreamTokenizer	307
	5.3.3.6. Clasa System	313
5.4.	Serializarea obiectelor	317
	5.4.1. Clasa ObjectOutputStream	317
	5.4.2. Clasa ObjectInputStream	320
	5.4.3. Interfața Serializable	322
	5.4.4. Interfața Externalizable	323
	5.4.5. Crearea unei clase serializabile	324
	5.4.5.1. Derivarea unei clase serializabile	324
	5.4.5.2. Implementarea interfeței Serializable	325
	5.4.5.3. Metode de serializare obișnuite	326
	5.4.5.4. Implementarea interfeței Externalizable	330
	5.4.5.5. Declararea interfeței și a tuturor metodelor de acces	331
5.5.	Concluzii	336
5.6.	Test-grila	336
5.7.	Exerciții propuse spre implementare	340
5.8.	Proiecte propuse spre implementare	341

6. FIRE DE EXECUȚIE	343
6.1. Cuvinte-cheie	343
6.2. Introducere	344
6.2.1. Breviar teoretic	344
6.2.2. Corectitudinea programelor	344
6.2.3. Primitivele de programare concurrentă	345
6.3. Programare concurrentă în Java	346
6.3.1. Crearea unui fir de execuție prin extinderea clasei Thread	347
6.3.2. Crearea unui <i>thread</i> utilizând interfața Runnable	348
6.3.3. Controlul unui fir de execuție	350
6.3.4. Prioritatea firelor de execuție	355
6.3.5. Grupurile de fire de execuție	359
6.3.6. Excluderea mutuală și sincronizarea	361
6.3.7. Metodele <code>wait()</code> și <code>notify()</code>	363
6.4. Studii de caz: problema filosofilor și problema producător-consumator	372
6.4.1. Problema filosofilor	372
6.4.2. Problema producător-consumator	377
6.5. Concluzii	382
6.6. Test-grilă	382
6.7. Exerciții propuse spre implementare	383
6.8. Proiecte propuse spre implementare	384
7. APPLETURI ȘI INSTRUMENTE DE LUCRU CU FERESTRELE (AWT)	385
7.1. Cuvinte-cheie	385
7.2. Appleturi	386
7.2.1. Clasa <code>Applet</code> (<code>java.applet.Applet</code>)	386
7.2.2. Obținerea parametrilor de intrare ai appleturilor	389
7.2.3. Contextul unui applet	391
7.2.4. Comunicarea dintre appleturi	392
7.2.5. Redarea sunetelor și imaginilor	394
7.2.5.1. Afișarea conținutului grafic	394
7.2.5.2. Redarea sunetelor	396
7.2.6. Transformarea unui applet într-o aplicație de sine stătătoare	398
7.3. Grafica în Java	399
7.4. Componente și evenimente	404
7.4.1. Bucla evenimentelor	404
7.4.2. Componente	408
7.4.3. Container	410
7.4.4. Etichete	410
7.4.5. Butoane	412
7.4.6. Câmpuri și arii text	414
7.4.6.1. Clasa <code>java.awt.TextComponent</code>	414
7.4.6.2. Clasa <code>java.awt.TextField</code>	415
7.4.6.3. Clasa <code>java.awt.TextArea</code>	416
7.4.7. Butoane de selecție (<code>checkbox</code> și <code>radio</code>)	421
7.4.8. Liste de opțiuni	423

7.4.9.	Suprafețe de desenare	428
7.4.10.	Panouri (<i>panels</i>)	429
7.4.11.	Meniuri <i>pop-up</i>	429
7.5.	Gestionari de poziționare (<i>Layout Managers</i>)	432
7.5.1.	Gestionarul <i>BorderLayout</i>	433
7.5.2.	Gestionarul <i>CardLayout</i>	436
7.5.3.	Gestionarul <i>FlowLayout</i>	438
7.5.4.	Gestionarul <i>GridLayout</i>	439
7.5.5.	Gestionarul <i>GridBagLayout</i>	440
7.5.6.	Poziționarea absolută	442
7.6.	Ferestre	444
7.6.1.	Clasa <i>java.awt.Window</i>	444
7.6.2.	Clasa <i>java.awt.Frame</i>	445
7.6.3.	Clasa <i>java.awt.Dialog</i>	447
7.6.4.	Clasa <i>java.awt.FileDialog</i>	449
7.7.	Managerul de securitate	451
7.8.	Concluzii	453
7.9.	Test-grila	453
7.10.	Exerciții propuse spre implementare	455
7.11.	Proiecte propuse spre implementare	457
8.	ACCESUL LA BAZE DE DATE FOLOSIND JDBC	459
8.1.	Cuvinte-cheie	459
8.2.	Introducere	460
8.3.	Clasificarea driverelor JDBC	462
8.4.	Breviar SQL	463
8.4.1.	Instrucțiunea <i>CREATE TABLE</i>	464
8.4.2.	Instrucțiunea <i>INSERT</i>	464
8.4.3.	Instrucțiunea <i>UPDATE</i>	465
8.4.4.	Instrucțiunea <i>DELETE</i>	465
8.4.5.	Instrucțiunea <i>SELECT</i>	466
8.5.	Asocieri de tipuri între SQL și Java	466
8.6.	Accesarea unei baze de date folosind JDBC	469
8.6.1.	Înregistrarea driverului JDBC folosind clasa <i>DriverManager</i>	469
8.6.2.	Stabilirea unei conexiuni cu baza de date	470
8.6.3.	Execuția unei instrucțiuni SQL	471
8.6.4.	Procesarea rezultatelor	474
8.6.5.	Închiderea unei conexiuni la o bază de date	478
8.7.	Instalarea și configurarea MySQL	479
8.7.1.	Introducere	479
8.7.2.	Driveri JDBC pentru MySQL	480
8.8.	Utilizarea punții JDBC-ODBC	483
8.9.	Tranzacții	486
8.9.1.	Motivații pentru folosirea tranzacțiilor	486
8.9.2.	ACID	488
8.9.3.	Niveluri de izolare	489
8.10.	Concluzii	490

8.11. Test-grilă	490
8.12. Exerciții propuse spre implementare	492
8.13. Proiecte propuse spre implementare	493
9. PROGRAMAREA REȚELOR	495
9.1. Cuvinte-cheie	495
9.2. Introducere	496
9.2.1. Adrese, porturi și socketuri	496
9.3. Programarea rețelor prin intermediul conexiunilor	499
9.3.1. Clasa <code>ServerSocket</code>	499
9.3.2. Clasa <code>Socket</code>	500
9.3.3. O aplicație simplă client/server orientată conexiune	502
9.4. Programarea rețelor prin intermediul datagramelor	507
9.4.1. Clasa <code>DatagramPacket</code>	507
9.4.2. Clasa <code>DatagramSocket</code>	509
9.4.3. O aplicație simplă client/server neorientată conexiune	511
9.4.4. Clasa <code>InetAddress</code>	514
9.4.5. Clasa <code>URL</code>	516
9.4.6. Obținerea unei pagini Web prin intermediul unui socket	522
9.5. Apelul metodelor la distanță	526
9.5.1. Obiectul de la distanță	527
9.5.2. Registrul de nume	529
9.5.3. Exemplu de utilizare a apelurilor la distanță	529
9.5.4. Localizarea fișierelor RMI. Pachete necesare	532
9.5.5. Clasa <code>Naming</code>	533
9.5.6. Clasa <code>LocateRegistry</code>	534
9.5.7. Interfața <code>Registry</code>	534
9.5.8. Clasa <code>RemoteObject</code>	535
9.5.9. Obiecte la distanță ca parametri	535
9.5.10. Serializarea unui obiect la distanță	537
9.5.11. Clasa <code>RemoteServer</code>	537
9.5.12. Clasa <code>RemoteStub</code>	538
9.5.13. Interfața <code>Unreferenced</code>	538
9.5.14. Clasa <code>RMIObjectFactory</code>	538
9.5.15. Interfața <code>RMIFailureHandler</code>	540
9.5.16. Exemplu de aplicație RMI	540
9.6. Concluzii	545
9.7. Test-grilă	545
9.8. Exerciții propuse spre implementare	547
9.9. Proiecte propuse spre implementare	547
10. SERVLETURI	549
10.1. Cuvinte-cheie	549
10.2. CGI (<i>Common Gateway Interface</i>)	550
10.2.1. Variabilele de mediu	551
10.2.2. Apelarea programelor CGI din formularele HTML	553

10.2.3. Procesarea datelor din formularele HTML prin metoda GET	555
10.2.4. Procesarea unui formular prin metoda POST	556
10.3. Servleturi	558
10.3.1. Servlet API	559
10.3.2. Funcționalitatea servleturilor	559
10.3.3. Execuția servleturilor	560
10.3.4. Structura de bază a unui servlet	561
10.3.5. Ciclul de viață al unui servlet	562
10.3.6. Clasa <code>HttpServlet</code>	564
10.3.7. Redirecțarea cererii	571
10.3.8. Returnarea unei erori	571
10.3.9. Exemplu de aplicație Web	572
10.4. Concluzii	583
10.5. Test-grila	584
10.6. Exerciții propuse spre implementare	585
10.7. Proiecte propuse spre implementare	585
11. JAVA SERVER PAGES (JSP)	587
11.1. Cuvinte-cheie	587
11.2. Introducere	588
11.3. Elemente JSP	589
11.3.1. Comentarii	590
11.3.2. Directive	591
11.3.2.1. <i>Directiva page</i>	592
11.3.2.2. <i>Directiva include</i>	593
11.3.2.3. <i>Directiva taglib</i>	594
11.3.3. Declarații	594
11.3.4. Inițializarea și terminarea unui JSP	595
11.3.5. Obiecte implicite	596
11.3.6. Expresii	597
11.3.7. Scriptlet-uri	597
11.3.8. Acțiuni	598
11.3.8.1. <i>Integrarea componentelor JavaBeans</i>	598
11.4. Concluzii	600
11.5. Test-grila	600
11.6. Exerciții propuse spre implementare	601
11.7. Proiecte propuse spre implementare	601
12. PROCESAREA DOCUMENTELOR XML	603
12.1. Cuvinte-cheie	603
12.2. Standarde și specificații	604
12.3. Instalare	605
12.4. SAX	605
12.4.1. Pachete necesare	605
12.4.2. Exemplu de document XML	605
12.4.3. Obținerea informațiilor din documentele XML	606

12.4.4.	Obținerea valorilor atributelor unui tag	610
12.4.5.	Procesarea documentelor XML care conțin spații de nume	611
12.4.6.	Procesarea documentelor XML care trebuie validate prin DTD	614
12.4.7.	Procesarea documentelor XML care trebuie validate via XML Schema	617
12.5.	DOM	619
12.5.1.	Pachete necesare	620
12.5.2.	Crearea arborelui asociat unui document XML	621
12.5.3.	Manipularea arborelui DOM	621
12.5.3.1.	<i>Interfața Node</i>	621
12.5.3.2.	<i>Procesarea documentelor XML care utilizează spații de nume</i>	628
12.5.3.3.	<i>Procesarea documentelor XML care conțin DTD</i>	631
12.5.3.4.	<i>Validarea documentelor XML prin XML Schema</i>	634
12.5.3.5.	<i>Interfața Document</i>	635
12.6.	XSLT	638
12.6.1.	Pachete necesare	638
12.6.2.	Realizarea unor transformări	639
12.7.	Concluzii	643
12.8.	Test-grila	644
12.9.	Exerciții propuse spre implementare	644
12.10.	Proiecte propuse spre implementare	645
13.	INTERACȚIUNEA CU UTILIZATORUL PRIN SWING	647
13.1.	Cuvinte-cheie	647
13.2.	Introducere	648
13.2.1.	Obiectivele urmărite în acest capitol	648
13.2.2.	Despre interfețe grafice	649
13.2.3.	JFC (<i>Java Foundation Classes</i>)	650
13.2.4.	Componentele și pachetele bibliotecii Swing	652
13.2.5.	Principii de bază	659
13.2.5.1.	<i>MVC (Model-View-Controller)</i>	659
13.2.5.2.	<i>UI Delegate</i>	660
13.2.5.3.	<i>Look-and-feel</i>	661
13.2.5.4.	<i>Mai multe despre modele</i>	664
13.2.5.5.	<i>Swing versus AWT</i>	667
13.3.	Fundamentele Swing	668
13.3.1.	Evenimente și ascultători	668
13.3.2.	Fire de execuție în Swing	681
13.3.3.	Desenarea componentelor grafice	684
13.3.4.	Alinierea la JavaBeans	689
13.3.5.	Gestiunea „focusului”	691
13.3.6.	Clasificarea componentelor Swing	695
13.4.	Containere de bază	695
13.4.1.	JFrame	696
13.4.2.	Clase și interfețe legate de ferestre	701
13.4.3.	JWindow	702
13.4.4.	JDialog	705
13.4.5.	JApplet	707
13.5.	Containere intermediare	708
13.5.1.	JPanel	708

13.5.2.	JScrollPane	709
13.5.3.	JSplitPane	713
13.5.4.	JTabbedPane	715
13.6.	Componente atomice simple	720
13.6.1.	JLabel	720
13.6.2.	Butoane	722
13.6.2.1.	JButton	724
13.6.2.2.	JToggleButton, JCheckBox, JRadioButton	724
13.6.3.	Chenare	727
13.6.4.	JList	730
13.6.5.	JComboBox	734
13.6.6.	JSpinner	738
13.7.	Componente atomice complexe	739
13.7.1.	Componente text	739
13.7.2.	JTable	748
13.7.3.	JTree	760
13.8.	Meniuri și bare de unelte	767
13.8.1.	Meniuri	767
13.8.2.	JToolBar	769
13.9.	Dialoguri	774
13.9.1.	JOptionPane	775
13.9.2.	JFileChooser	780
13.9.3.	JColorChooser	785
13.10.	Componente pentru progres și derulare	789
13.10.1.	JSlider	789
13.10.2.	JScrollBar	791
13.10.3.	JProgressBar	793
13.10.4.	JToolTip	795
13.11.	Ferestre interioare	796
13.11.1.	JDesktopPane	797
13.11.2.	JInternalFrame	797
13.12.	Recomandări și optimizări	802
13.13.	Concluzii	803
13.14.	Test-grilă	804
13.15.	Exerciții propuse spre implementare	809
13.16.	Proiecte propuse spre implementare	810
14.	SISTEMUL HELP PENTRU JAVA	811
14.1.	Cuvinte-cheie	811
14.2.	Convenții	812
14.3.	Principii de bază	812
14.3.1.	Prezentarea pachetului JavaHelp	813
14.3.2.	Alcătuirea ferestrei standard JavaHelp	814
14.3.3.	Principii pe care sistemul JavaHelp le respectă	815
14.4.	Dezvoltarea efectivă a help-ului	815
14.4.1.	Temele HTML	816
14.4.2.	Fișierul HelpSet	817

SQL permite și definirea de tipuri de date utilizator, cunoscute și sub numele de UDT (engl. *User Defined Types*) prin intermediul instrucțiunilor SQL `CREATE TYPE`. Acestea se împart în tipuri structurate și tipuri distincte. Spre exemplu, prezentăm modul de creare a unui tip structurat de date:

```
CREATE TYPE PUNCT_PLAN
(
    X FLOAT,
    Y FLOAT
)
```

Un tip distinct poate fi creat având la bază un tip deja existent, așa cum se poate vedea în exemplul următor:

```
CREATE TYPE NUMERE AS NUMERIC(10, 2)
```

Prin definiție, un tip distinct SQL este asociat aceleiași tip ca și tipul de bază. În exemplul prezentat, deoarece tipul `NUMERIC` este asociat cu `java.math.BigDecimal`, aceeași asociere va căpăta și tipul `NUMERE`. Deci vom volosi `ResultSet.getBigDecimal()` pentru a obține o valoare de acest tip.

8.6. Accesarea unei baze de date folosind JDBC

Procesul obținerii de informații dintr-o bază de date folosind JDBC implică în principiu cinci pași:

1. înregistrarea driverului JDBC folosind gestionarul de drivere `DriverManager`;
2. stabilirea unei conexiuni cu baza de date;
3. execuția unei instrucțiuni SQL;
4. procesarea rezultatelor;
5. închiderea conexiunii cu baza de date.

8.6.1. Înregistrarea driverului JDBC folosind clasa `DriverManager`

Rolul managerului de drivere este acela de a ține o referință la fiecare dintre obiectele driver disponibile în aplicația curentă. Un driver JDBC este înregistrat automat de managerul de drivere atunci când clasa driver este încărcată dinamic. Pentru încărcarea dinamică a unui driver JDBC, folosim metodele `Class.forName()`. După cum se observă, nu este nevoie de crearea unei instanțe a clasei driver odată ce aceasta a fost încărcată. Dacă se apelează metoda `newInstance()`, se va realiza un duplicat nefolositor al clasei driver. În exemplul următor vom încărca driverul punte JDBC-ODBC din pachetul `sun.jdbc.odbc` și, mai apoi, driverul `MySQL Connector/J`, care permite conectarea la serverul de baze de date `MySQL`. Trebuie remarcat faptul că pentru `Connector/J` este necesar să includem în `CLASSPATH` calea spre pachetul care conține driverul.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("com.mysql.jdbc.Driver");
```

`Class.forName()` este o metodă statică. Aceasta permite mașinii virtuale Java să aloce dinamic, să încarce și să facă o legătură la clasa specificată ca argument printr-un șir de caractere. În cazul în care clasa nu este găsită, se aruncă o excepție `ClassNotFoundException`. A se vedea `Java Reflection API`.

Drivererele pot fi înregistrate și folosind metoda `DriverManager.registerDriver()`.

8.6.2. Stabilirea unei conexiuni cu baza de date

Odată ce s-a încărcat un driver, putem să-l folosim pentru stabilirea unei conexiuni cu baza de date. O conexiune JDBC este identificată printr-un URL JDBC specific. Sintaxa standard pentru URL-ul unei baze de date este:

```
jdbc:<subprotocol>:<nume>
```

Prima parte precizează că pentru stabilirea conexiunii se folosește JDBC. Partea de mijloc `<subprotocol>` este un nume de driver valid sau al altei soluții de conexiune a bazelor de date. Ultima parte, `<nume>`, este un nume logic sau alias care corespunde bazei de date fizice. Dacă baza de date va fi accesată prin internet, secțiunea `<nume>` va respecta următoarea convenție de nume:

```
//numegazda:port/subsubnume
```

În acest sens, un exemplu de adresă corectă este:

```
jdbc:mysql://localhost:386/arhiva
```

Prezentăm în continuare sintaxa particulară pentru câteva drivere JDBC:

ODBC - `jdbc:odbc:<sursa_date>[:<nume_tribut>=<valoare_tribut>]*`

MySQL - `jdbc:mysql://server[:port]/numeBazaDate`

Oracle - `jdbc:oracle:thin:@server:port:numeInstanta`

Pentru stabilirea unei conexiuni la o bază de date, se folosește metoda statică `getConnection()` din clasa `DriverManager`.

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost/arhiva");
```

Pentru baze de date care necesită autentificare, se utilizează o formă a acestei metode cu trei argumente.

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost/arhiva", nume_utilizator, parola);
```

Pentru a afla informații despre DBMS, va trebui să apelăm `Connection.getMetaData()` pentru a obține o instanță `DatabaseMetaData`, căreia îi putem aplica diverse metode de a afla informații despre tabellele bazei de date, sintaxa SQL suportată, dacă suportă sau nu proceduri stocate etc.

Folosind JNDI (*Java Name and Directory Interface*) putem să ne conectăm la o bază de date considerând-o sursă de date având un nume logic, în loc să codăm direct în aplicație numele ei și driverul pe care îl vom folosi. Acest mod de conectare nu este obiectul acestui capitol, fiind o caracteristică a sistemelor distribuite.

8.6.3. Execuția unei instrucțiuni SQL

După ce s-a stabilit conexiunea, se pot trimite instrucțiuni SQL către baza de date. API-ul JDBC nu verifică dacă instrucțiunea este corectă și nici apartenența ei la un anumit standard SQL, permițându-se astfel trimiterea chiar de instrucțiuni nonSQL. Programatorul este cel care știe dacă DBMS-ul interogată suportă interogările pe care le trimite și, dacă nu, el este cel care va trata excepțiile primite drept răspuns. Dacă instrucțiunea este SQL, atunci aceasta poate face anumite operații asupra bazei de date, cum ar fi căutare, inserare, actualizare sau ștergere.

API-ul JDBC specifică trei interfețe (pe care dezvoltatorul driverului trebuie să le implementeze) pentru trimiterea de interogări către bazele de date, fiecareia corespunzându-i o metodă specială în clasa `Connection`, de creare a instanțelor corespunzătoare. Acestea sunt prezentate în tabelul care urmează:

Clasa	Metoda de creare	Explicații
<code>Statement</code>	<code>Connection.createStatement()</code>	Este folosită pentru trimiterea de instrucțiuni SQL simple, fără parametri.
<code>PreparedStatement</code>	<code>Connection.prepareStatement()</code>	Permite folosirea instrucțiunilor SQL precompilate și a parametrilor de intrare în interogări. Această metodă de a face interogări este utilă în cazul în care se fac interogări care diferă doar printr-un număr de parametri.
<code>CallableStatement</code>	<code>Connection.prepareCall()</code>	Permite folosire procedurilor stocate pe serverul DBMS.

Pentru execuția unei instrucțiuni SQL neparametrizate, se folosește metoda `createStatement()`, aplicată unui obiect `Connection`. Această metodă întoarce un obiect din clasa `Statement`.

```
Statement instructiune = con.createStatement();
```

Putem aplica apoi una dintre metodele `executeQuery()`, `executeUpdate()` sau `execute()` obiectului de tip `Statement` pentru a trimite DBMS-ului instrucțiunile SQL. Metoda `executeQuery()` este folosită în cazul interogărilor care returnează mulțimi-rezultat (instanțe ale clasei `ResultSet`), așa cum este cazul instrucțiunilor `SELECT`. Pentru operațiile de actualizare sau ștergere, cum ar fi `INSERT`, `UPDATE` sau `DELETE`, se folosește metoda `executeUpdate()` aplicată obiectului de tip `Statement`, rezultând un întreg care reprezintă numărul înregistrării afectate. Aceeași metodă este folosită pentru interogările SQL DDL, cum ar fi `CREATE TABLE`, `DROP TABLE` și `ALTER TABLE`, în acest caz returnând întotdeauna zero. Metoda `execute()` este

utilizată în cazul în care se obține mai mult de o mulțime-rezultat sau un număr de linie.

```
ResultSet rs = instructiune.executeQuery(
    "select * from arhive");
String sql = "insert into arhive values (
    'Popescu', 'Ion', 'Iasi')";
int raspuns = instructiune.executeUpdate(sql);
```

JDBC 2.0 permite trimiterea spre execuție a mai multor instrucțiuni SQL grupate (engl. *batch*), această facilitare mărind uneori performanțele. Prezentăm în continuare un astfel de caz:

```
Statement inter = con.createStatement();
con.setAutoCommit(false);

inter.addBatch("INSERT INTO arhive VALUES (1, 'Popescu', 'Ioan')");
inter.addBatch("INSERT INTO cantitati VALUES (260, 'file')");
inter.addBatch("INSERT INTO localitati VALUES ('iasi', 'oras')");
int [] actualizari = inter.executeBatch();
```

Se observă dezactivarea modului *autocommit*. Acest aspect determină ca modificările rezultate în tabele după execuția metodei `executeBatch()` să nu fie automat salvate permanent (engl. *commit*) sau anulate (engl. *rollback*), aceste operațiuni rămânând la alegerea clientului. Se poate astfel ca în cazul în care una dintre interogări eșuează, clientul să anuleze efectele tuturor. Pentru alte informații, puteți consulta secțiunea dedicată tranzacțiilor. Metoda `executeBatch()` returnează un tablou în care elementele corespund interogărilor SQL efectuate și reprezintă numărul de linii afectate de instrucțiunile SQL corespunzătoare.

Pentru a executa independent instrucțiunile SQL, se poate păstra modul *autocommit* folosind captarea excepțiilor `BatchUpdateException` aruncate în caz de eroare, așa cum se poate vedea în exemplul următor:

```
try {
    inter.addBatch("INSERT INTO arhive VALUES (1, 'Popescu', 'Ioan')");
    inter.addBatch("INSERT INTO cantitati VALUES (260, 'file')");
    inter.addBatch("INSERT INTO localitati VALUES ('iasi', 'oras')");
    int [] actualizari = stmt.executeBatch();
} catch (BatchUpdateException b) {
    System.err.println("Actualizari realizate: ");
    int [] eroriActualizari = b.getUpdateCounts();
    for (int i = 0; i < eroriActualizari.length; i++) {
        System.err.print(eroriActualizari[i] + " ");
    }
    System.err.println("");
}
```

Ștergerea comenzilor dintr-un grup se realizează cu metoda `clearBatch()`.

Un driver JDBC poate să nu ofere suport pentru execuția grupată a unor instrucțiuni SQL. Pentru a afla dacă e permis acest lucru, se va folosi metoda `supportsBatchUpdates()` din clasa `DatabaseMetaData`.

Dacă se dorește realizarea de apeluri SQL având date variabile drept intrare, se va folosi clasa `PreparedStatement` care moștenește clasa `Statement`. Prezentăm în continuare modul de construcție a unei astfel de instrucțiuni, unde `con` reprezintă o instanță `Connection`:

```
PreparedStatement instructiune = con.prepareStatement(  
    "update arhive set nume = ? where prenume like ?");
```

După cum se observă, prin construcție, obiectul de tip `PreparedStatement` primește ca intrare o instrucțiune SQL (spre deosebire de cazul `Statement`). În momentul construcției, instrucțiunea SQL primită ca parametru este trimisă direct spre DBMS, unde este precompilată. Mai rămâne să dăm valori variabilelor folosind metode `setXX()` corespunzătoare tipurilor de date și să executăm efectiv interogarea, după cum se poate vedea în continuare:

```
instructiune.setString(1, "Popescu");  
instructiune.setString(2, "Ion");  
instructiune.executeUpdate();
```

Această manieră de interogare este mai rapidă decât folosirea clasei `Statement`, în cazul în care dorim execuția repetată a unei instrucțiuni SQL. Instrucțiunea SQL este compilată o singură dată în momentul creării instanței clasei `PreparedStatement` și folosită apoi repetat, eventual cu valori diferite ale parametrilor. Se poate folosi și în cazul în care nu avem parametri, după cum se observă în continuare:

```
PreparedStatement instructiune = con.prepareStatement(  
    "select * from arhive");  
ResultSet rs = instructiune.executeQuery();
```

Nu vom prezenta în acest capitol procedurile stocate datorită nivelului relativ ridicat de complexitate.

Metodele fără parametri de intrare puse la dispoziție de clasa `Connection` pentru cele trei interfețe prezentate în tabelul precedent produc mulțimi-rezultat fără cursor deplasabil și nesenzitive la modificări. O mulțime-rezultat are cursor pentru poziționare deplasabil, proprietate care poartă numele de *scrolling*, dacă deține un pointer interior care indică înregistrarea accesată la momentul curent, pointer ce se poate deplasa programatic. De asemenea, după cum spuneam, mulțimea-rezultat nu este senzitivă (engl. *sensitive*) la modificările care pot surveni între timp în tabela interogată. Prin senzitivitate se înțelege actualizarea automată a mulțimii-rezultat pentru a ilustra dinamic modificările survenite între timp în tabelă.

Driverurile care sunt conforme cu versiunea 2.0 a specificației JDBC permit obținerea de mulțimi-rezultat senzitive și având cursor deplasabil. Acest lucru este posibil prin specificarea în constructorul instanței `Statement` a uneia dintre constantele prezentate în tabelul care urmează:

Constantă	Explicații
TYPE_FORWARD_ONLY	Acest tip de <code>ResultSet</code> este cel din JDBC 1.0. Mulțimea-rezultat nu are cursor deplasabil decât dinspre început spre sfârșit. Modificările făcute în tabelă nu se reflectă în mulțimea-rezultat.
TYPE_SCROLL_INSENSITIVE	Mulțime-rezultat <i>scrollable</i> , deci cursorul poate fi mutat înainte și înapoi sau pe orice poziție diferită de poziția curentă. De asemenea, eventualele modificări făcute între timp în tabelă nu sunt reflectate în <code>ResultSet</code> .
TYPE_SCROLL_SENSITIVE	Mulțime-rezultat <i>scrollable</i> . Sensitivitate la modificări.

De asemenea, driverule conform cu specificația 2.0 a JDBC oferă și posibilitatea actualizării programatice a tabelor prin intermediul obiectelor `ResultSet`. Tipul de actualizare se specifică prin intermediul uneia dintre constantele prezentate în continuare:

Constantă	Explicații
CONCUR_READ_ONLY	<code>ResultSet</code> -ul nu poate fi modificat programatic. Oferă posibilitatea unui număr nelimitat de conectări concurente la baza de date, deoarece nu se fac modificări asupra tabelii care ar putea genera conflicte. Acest tip de <code>ResultSet</code> este specific driverelor conforme specificației JDBC 1.0.
CONCUR_UPDATABLE	<code>ResultSet</code> -ul și baza de date pot fi modificate programatic. Sunt posibile un număr limitat de conexiuni concurente dat de tipul de concurență ales (a se vedea secțiunea dedicată tranzacțiilor). Acest tip de <code>ResultSet</code> este specific driverelor conforme specificației JDBC 2.0.

Prezentăm un exemplu de cod care va determina crearea unei mulțimi-rezultat având cursor deplasabil și sensibil la modificări (ordinea parametrilor metodei `createStatement()` este importantă).

```
Statement declar = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.
    CONCUR_UPDATABLE);
// la fel se poate crea o instanța a clasei PreparedStatement
ResultSet rez = declar.executeQuery(
    "select nume, prenume from arhive");
```

8.6.4. Procesarea rezultatelor

Pentru parcurgerea simplă a înregistrărilor unui obiect din clasa `ResultSet` folosind JDBC 1.0, putem folosi metoda `next()`, ca în următoarea secvență de cod.

```
while (rs.next()) // implicit, cursor poziționat înainte de prima
```